UNIT 3
INTERMEDIATE CODE GENERATION

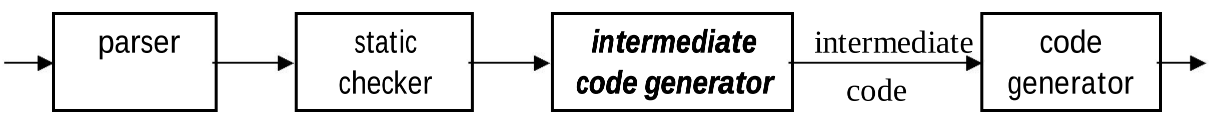
3.1 INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



3.2INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

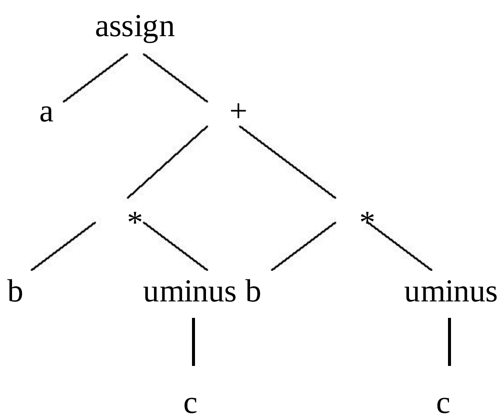
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

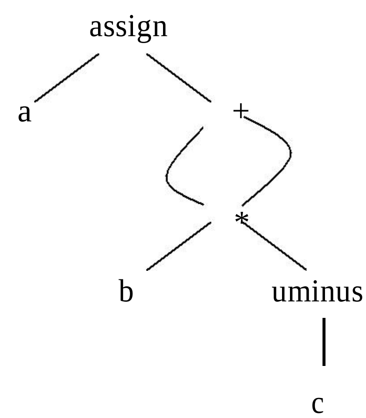
Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag(Directed Acyclic Graph)** gives the same information but in a more compact way becausecommon subexpressions are identified. A syntax tree and dag for the assignment statement **a : =b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

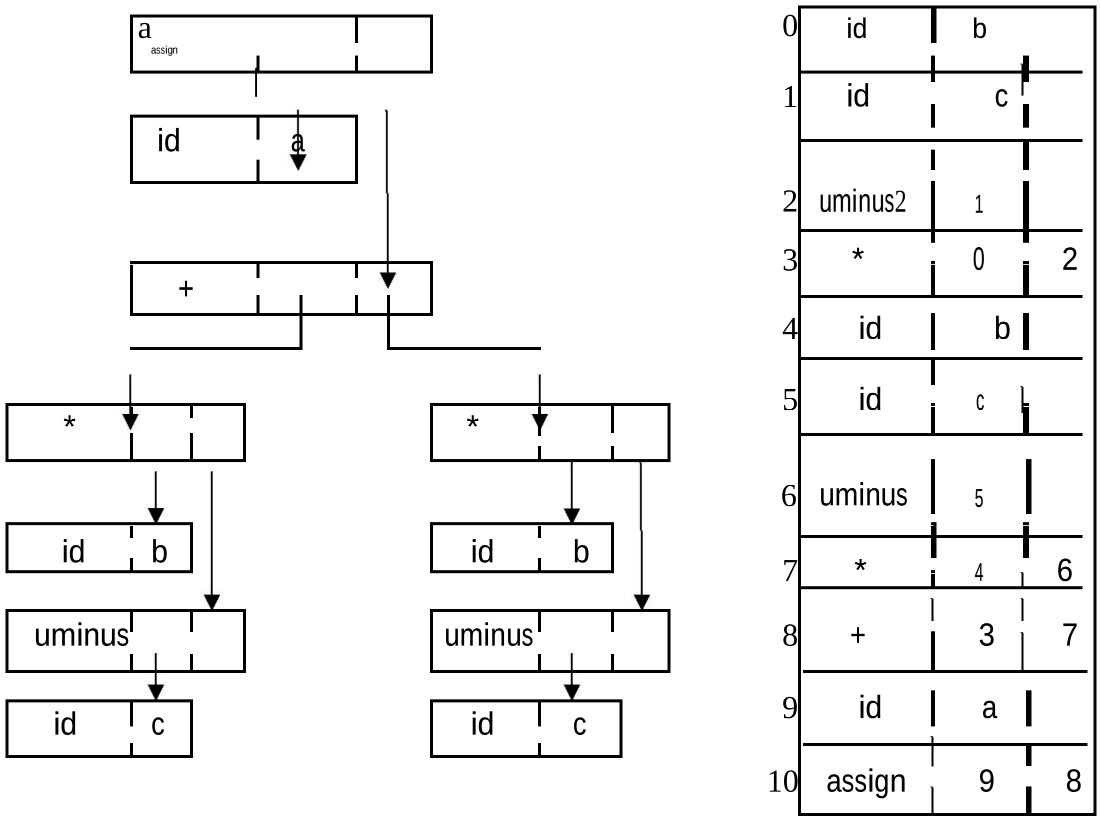
PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 +E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



(a) (b)

Three-Address Code:

Three-address code is a sequence of statements of the general form

x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into asequence

t_1 : = y * z t_2 :
= x + t_1

wheret_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

(a) Code for the syntax tree	(b) Code for the dag
t_1 : = - c	t_1 : = -c
t_2 : = b * t_1	t_2 : = b * t_1
t_3 : = - c	t_5 : = t_2 + t_2
t_4 : = b * t_3	a : = t_5
t_5 : = t_2 + t_4	
a : = t_5	

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three -Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x : = y*opz***, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : =*op*y**, where ***op*** is a unary operation. Essential unary

operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. *Copy statements* of the form**x : = y**where the value of *y* is assigned to *x*.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as **if*x relop y*goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation *relop to y*. If not, the three-address statement following if *x relop y* goto L is executed next,as in the usual sequence.

6. *param x* and *call p, n* for procedure calls and *return y,* where y representing a returned valueis optional. For example,

   param
   x_1
   param
   x_2
   . . .
   param
   x_n call
   p,n

   generated as part of a call of the procedure p(x_1, x_2, …. ,x_n).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

3.3 SYNTAX-DIRECTED TRANSLATION INTO THREE-ADDRESS CODE:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : =*E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

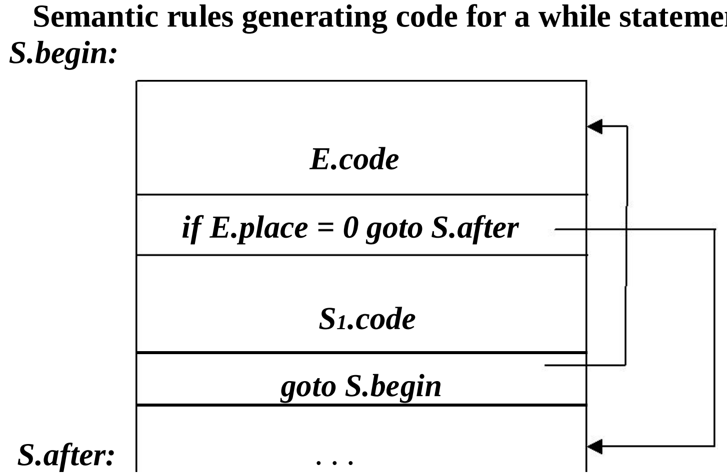
Given input a : = b * - c + b * - c, the three-address code is as shown above.
The synthesized attribute *S.code* represents the three-address code for the assignment *S.* The nonterminal *E* has two attributes :

1. *E.place*, the name that will hold the value of E , and
2. *E.code*, the sequence of three-address statements evaluating E.

Syntax -directed definition to produce three-address code for assignments

PRODUCTION	**SEMANTIC RULES**
S → id : = E	***S.code : = E.code || gen(id.place ':=' E.place)***
E → E_1 + E_2	***E.place := newtemp;*** ***E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)***
E → E_1 * E_2	***E.place := newtemp;*** ***E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)***
E → - E_1	***E.place := newtemp;*** ***E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)***
E → (E_1)	***E.place : = E_1.place;*** ***E.code : = E_1.code***
E → id	***E.place : = id.place;*** ***E.code : = ' '***

Semantic rules generating code for a while statement



PRODUCTION	**SEMANTIC RULES**
S → while *E* do *S_1*	***S.begin := newlabel;*** ***S.after := newlabel;*** ***S.code := gen(S.begin ':') ||*** ***E.code ||*** ***gen ('if' E.place '=' '0' 'goto' S.after)||*** ***S_1.code ||*** ***gen ('goto' S.begin) ||*** ***gen (S.after ':')***

- The function *newtemp* returns a sequence of distinct names $t_1,t_2,\ldots$.. in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x*, *y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S→*** **while *E* do *S₁*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

3.4 IMPLEMENTATION OF THREE-ADDRESS STATEMENTS:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : =y op z** is represented by placing*y*in*arg1*,*z*in*arg2*and*x*in*result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three
- The fields *arg1* and *arg2,* for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).

- Since three fields are used, this intermediate code format is known as *triples*.

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above:
A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

(b) x : = y[i]

	op	*arg1*	*arg2*
(0)	= [ ]	y	I
(1)	assign	x	(0)

- fields: *op, arg1* and *arg2.*

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

3.5 DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id :*T*.**
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter(name, type, offset*) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types*integer* and *real* by applying the type constructors pointer and array. If type expressionsarerepresented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

$P \rightarrow D$	{ *offset* : = 0 }
$D \rightarrow D \,;\, D$	
$D \rightarrow id : T$	{ *enter(id.name, T.type, offset)*; *offset* : = *offset* + *T.width* }
$T \rightarrow integer$	{ *T.type* : = *integer*; *T.width* : = 4 }
$T \rightarrow real$	{ *T.type* : = *real*; *T.width* : = 8 }
$T \rightarrow array\ [\ num\ ]\ of\ T_1$	{ *T.type* : = *array(num.val,* T_1*.type)*; *T.width* : = *num.val* X T_1*.width* }
$T \rightarrow \uparrow T_1$	{ *T.type* : = *pointer* (T_1*.type)*; *T.width* : = 4 }

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

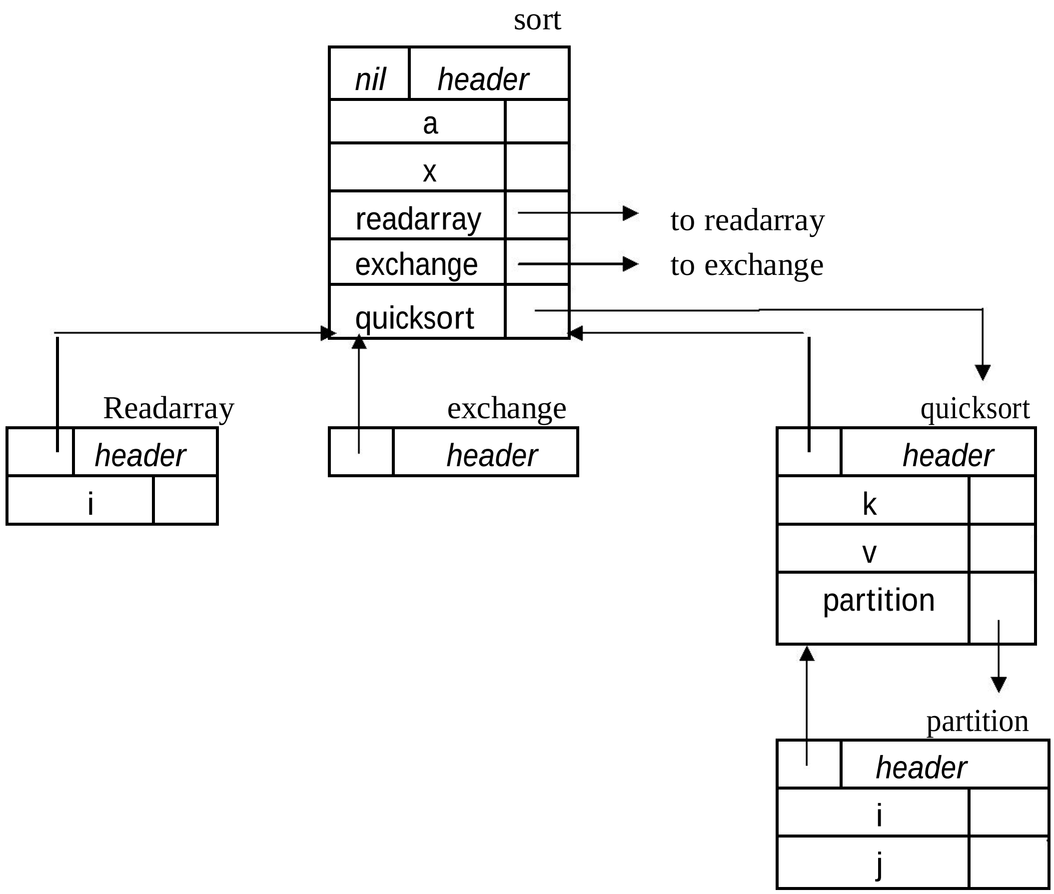
$D \rightarrow D \,;\, D$ | **id** : T | **proc id** ; D ; S

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow$ ***proc id*** $D_1;S$ is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray*,exchange, and *quicksort* pointing back to that for the containing procedure sort, consisting ofthe entireprogram. Since *partition* is declared within *quicksort*, its table points to that of*quicksort*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. Theargument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointedto by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.
3. *addwidth(table, width)* records the cumulative width of all the entries in table in the headerassociated with this symbol table.
4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol tablepointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

P → M D ***{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }***

M → ε ***{ t : = mktable (nil); push (t,tblptr); push (0,offset) }***

D → D₁ ; D₂

D → proc id ; N D₁ ; S ***{ t : = top (tblptr); addwidth (t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) }***

D → id : T ***{ enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width }***

N → ε ***{ t := mktable (top (tblptr)); push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.
- The top element of stack *offset* is the next available relative address for a local of the current procedure.
- All semantic actions in the subtrees for B and C in

  A → BC {$action_A$}

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.
- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil)*. The action also pushes relative address 0 onto stack offset.
- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.
- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.
- When the action on the right side of *D→proc id; ND₁; S* occurs, the width of all

declarations generated by D_1 is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

3.6 ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

P → M D

M → ε

D → D ; D | **id** : T | **proc id** ; N D ; S

N → ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

```
S → id : =  E   { p : = lookup ( id.name);
                  if p ≠nil then
                  emit( p ‘ : =’ E.place)
                  else error }

E → E1 + E2    { E.place : = newtemp;
                  emit( E.place ‘: =’ E1.place ‘ + ‘ E2.place ) }

E → E1 * E2    { E.place : = newtemp;
                  emit( E.place ‘: =’ E1.place ‘ * ‘ E2.place ) }

E →- E1        { E.place : = newtemp;
                  emit ( E.place ‘: =’ ‘uminus’ E1.place ) }

E → ( E1 )     { E.place : = E1.place }

E → id         { p : = lookup ( id.name);

                  if p ≠nil then
                       E.place : =
                  p else error
               }
```

Reusing Temporary Names

- The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
- Temporaries can be reused by changing *newtemp*. The code generated by the rules for E → $E_1 + E_2$ has the general form:

  evaluate E_1 into t_1
  evaluate E_2 into t_2
  $t := t_1 + t_2$

- The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1.
- For example, consider the assignment x := a * b + c * d – e * f

Three-address code with stack temporaries

statement	*value of c*
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *w*, then the *i*th element of array A begins in location

$$base + (i - low) \times w$$

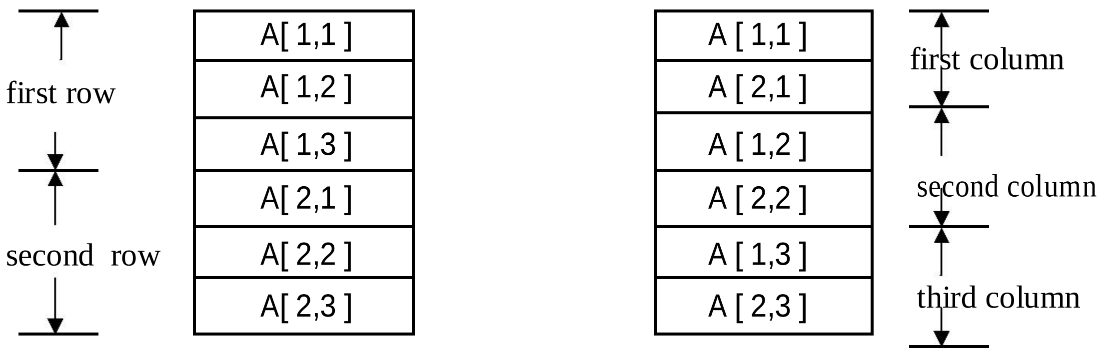
where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of A*[low]*.
The expression can be partially evaluated at compile time if it is rewritten as

$$i \times w + (base - low \times w)$$

The subexpression ***c = base – lowxw*** can be evaluated when the declaration of the array is seen. Let us assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding *i* x *w* to *c*.

Address calculation of multi-dimensional arrays:
A two-dimensional array is stored in of the two forms :
- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR **(b) COLUMN-MAJOR**

In the case of row-major form, the relative address of A[i_1 ,i_2] can be calculated by the formula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2, then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of A[$i_1,i_2,\ldots,i_k$]

$((\ldots ((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w + base - ((\ldots ((low_1 n_2 + low_2) n_3 + low_3) \ldots) n_k + low_k) \times w$

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist$]
(6) $L \rightarrow$ **id**
(7) $Elist \rightarrow Elist , E$
(8) $Elist \rightarrow$ **id**[E

Let us generate a normal assignment if *L* is a simple name, and an indexed assignment into the location denoted by *L* otherwise :

(1) $S \rightarrow L := E$ { **if** *L.offset* = **null then** / * *L is a simple* **id** */
emit (*L.place* ‘: =’ *E.place*);
else
emit (*L.place* ‘ [‘ *L.offset* ‘]’ ‘: =’*E.place*) }

(2) $E \rightarrow E_1 + E_2$ { *E.place* : = *newtemp;*
emit (*E.place* ‘: =’ E_1*.place* ‘ +’ E_2*.place*)}

(3) $E \rightarrow (E_1)$ { *E.place* : = E_1*.place*}

When an array reference *L* is reduced to *E* , we want the *r*-value of *L*. Therefore we use indexing to obtain the contents of the location *L.place* [*L.offset*] :

(4) $E \rightarrow L$ { **if** *L.offset* = **null then** /* *L is a simple* **id** * /
E.place : = *L.place*
else begin
E.place : = *newtemp;*
emit (*E.place* ‘: =’ *L.place* ‘ [‘ *L.offset* ‘]’)
end }

(5) $L \rightarrow Elist$] { *L.place* : = *newtemp;*
L.offset : = *newtemp;*
emit (*L.place* ‘: =’ *c*(*Elist.array*));
emit (*L.offset* ‘: =’ *Elist.place* ‘*’ *width* (*Elist.array*)) }

(6) $L \rightarrow$ **id** { *L.place* := **id**.*place;*

L.offset := **null**}

(7) $Elist \rightarrow Elist_1 , E$ { *t* := *newtemp;*
m : = $Elist_1.ndim$ + 1*;*
emit (*t* ': =' $Elist_1.place$ '*' *limit*
($Elist_1.array,m$)); *emit* (*t* ': =' *t* '+' *E.place*);
Elist.array : = $Elist_1.array$;
Elist.place : = *t;*
Elist.ndim : = *m* }

(8) $Elist \rightarrow$ **id** [*E* { *Elist.array* : =**id**.*place;*
Elist.place : = *E.place;*
Elist.ndim : = 1 }

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production $E \rightarrow E + E$ is :

$E \rightarrow E + E$ { *E.type* : =
if $E_1.type$ = *integer* **and**
$E_2.type$ = *integer* **then** *integer*
else *real*}

The entire semantic rule for $E \rightarrow E + E$ and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to areal of equal value, called x.

Semantic action for $E \rightarrow E_1 + E_2$

E.place := *newtemp;*
if $E_1.type$ = *integer and* $E_2.type$ = *integer then*
beginemit(E.place ': =' $E_1.place$ 'int +'
$E_2.place$); *E.type* : = *integer*
end
else if $E_1.type$ = *real and* $E_2.type$ = *real then*
beginemit(E.place ': =' $E_1.place$ 'real +'
$E_2.place$); *E.type* : = *real*
end
else if $E_1.type$ = *integer* **and** $E_2.type$ = *real* **then**

begin *u : = newtemp;*
emit(u ': =' 'inttoreal' *E_1.place); emit(*
E.place ': =' u ' real +' *E_2.place);*
E.type : = real
end
else if *E_1.type = real* **and** *E_2.type =integer* **then**
begin *u : = newtemp;*
emit(u ': =' 'inttoreal' *E_2.place*); *emit(*
E.place ': =' E_1.place ' real +' *u);*
E.type : = real
end
else
E.type : = type_error;

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

t_1 : = i int* j t_3 :
= inttoreal t_1 t_2 :
= y real+ t_3 x : =
t_2

3.7 BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form *E_1***relop***E_2*, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E → E **or** E | E **and** E | **not** E | (E) | **id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly

convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for

  a **or** b **and not** c is

  the three-address sequence

  t_1 : = **not** c t_2

  : = b **and** t_1 t_3

  : = a **or** t_2

- A relational expression such as a < b is equivalent to the conditional

  statement if a < b then 1 else 0

  which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

  100 : if a < b goto 103
  101 : t : = 0
  102 : goto 104
  103 : t : = 1
  104 :

Translation scheme using a numerical representation for booleans

E → *E*$_1$ **or** *E*$_2$	*{ E.place : = newtemp;* *emit(E.place ': =' E*$_1$*.place '***or***'E*$_2$*.place)}*
E → *E*$_1$ **and** *E*$_2$	*{ E.place : = newtemp;* *emit(E.place ': =' E*$_1$*.place '***and***'E*$_2$*.place)}*
E → **not** *E*$_1$	*{ E.place : = newtemp;* *emit(E.place ': =' '***not***' E*$_1$*.place)}*
E → *(E*$_1$ *)*	*{ E.place : = E*$_1$*.place }*
E → **id**$_1$ **relop id**$_2$	*{ E.place : = newtemp;* *emit('if'* **id**$_1$*.place* **relop***.op* **id**$_2$*.place '***goto***' nextstat +***3***);* *emit(E.place ': =' '***0***');* *emit('***goto***' nextstat +***2***);* *emit(E.place ': =' '***1***') }*
E → **true**	*{ E.place : = newtemp;* *emit(E.place ': =' '***1***') }*
E → **false**	*{ E.place : = newtemp;* *emit(E.place ': =' '***0***') }*

Short-Circuit Code:

It is also possible to translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and,or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto 103	107 : t_2 : = 1
101 : t_1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t_3 : = 0
103 : t_1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t_3 : = 1
105 : t_2 : = 0	112 : t_4 : = t_2 and t_3
106 : goto 108	113 : t_5 : = t_1 or t_4

Flow-of-Control Statements

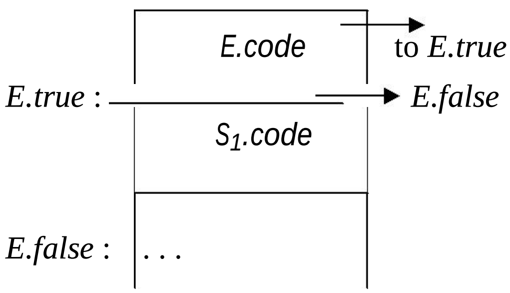
Let us consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S →**if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

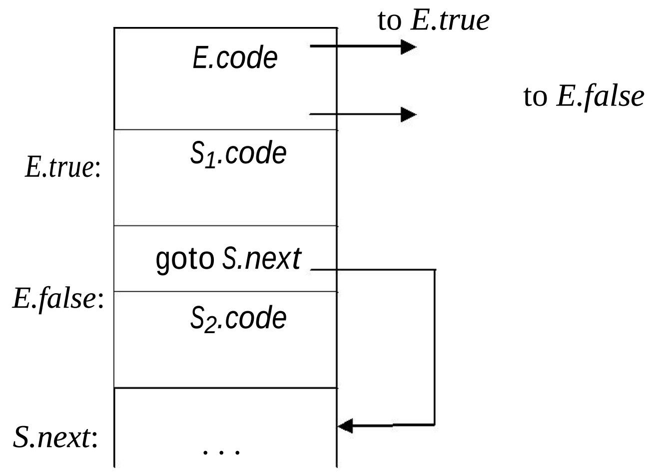
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

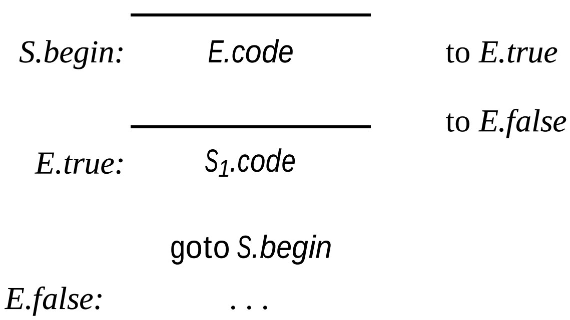
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen(\text{'}\mathbf{goto}\text{'}\ S.next) \mid\mid$ $gen(E.false\ ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$ $gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen(\text{'}\mathbf{goto}\text{'}\ S.begin)$

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.false\ ':') \mid\mid E_2.code$
$E \rightarrow E_1$ **and** E_2	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$

	$E.code := E_1.code \parallel gen(E_1.true\ ':') \parallel E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow$ **id**$_1$**relop id**$_2$	$E.code := gen($'**if**'**id**$_1.place$ **relop.**op **id**$_2.place$ '**goto**' $E.true) \parallel gen($'**goto**' $E.false)$
$E \rightarrow$ **true**	$E.code := gen($'**goto**' $E.true)$
$E \rightarrow$ **false**	$E.code := gen($'**goto**' $E.false)$

3.8 CASE STATEMENTS

The "switch" or "case" statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

switch *expression*
    begin
        case *value*:*statement*
        case *value*:*statement*
            . . .
        case *value*:*statement*
        default : *statement*
    end

There is a selector expression, which is to be evaluated, followed by *n* constant values that the expression might take, including a default "value" which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.

- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of i_{max} - i_{min}, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - i_{min} and the label for the default in entries not filled otherwise. To perform switch,

evaluate the expression to obtain the value of *j* , check the value is within range and transfer to the table entry at offset j-i_{min} .

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```
switch E begin
   case V1:S1 case V2:S2
                ...
          case Vn-1:Sn-1
          default :  Sn
      end
```

This case statement is translated into intermediate code that has the following form :

Translation of a case statement

```
            code to evaluate E into t
            goto test
L1:         code for S1
            goto next
L2:         code for S2
            goto next
              ...
Ln-1:       code for Sn-1
            goto next
Ln:         code for Sn
            goto next
test :      if  t = V1 goto L1
            if  t = V2 goto L2
              ...
            if  t = Vn-1 goto Ln-1
next :      goto Ln
```

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression E is parsed, the code to evaluate E into **t** is generated. After processing E , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label L_i is created and entered into the symbol table. A pointer to this symbol-table entry and the value V_i of case constant are placed on a stack (used only to store cases).
- Each statement **case**V_i: S_i is processed by emitting the newly created label L_i, followed by the code for S_i , followed by the jump **goto next**.
- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

  case V_1 L_1
  case V_2 L_2
  . . .
  case V_{n-1} L_{n-1}
  case t L_n
  label next

  where t is the name holding the value of the selector expression E, and L_n is the label for the default statement.

3.9 BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels ***backpatching***.

To manipulate lists of labels, we use three functions :

1. *makelist*(i) creates a new list containing only i, an index into the array*of* quadruples; *makelist* returns a pointer to the list it has made.
2. *merge*(p_1,p_2) concatenates the lists pointed to by $p_1$and p_2, and returns a*pointer* to the concatenated list.

3. *backpatch(p,i)* inserts i as the target label for each of the statements on the list pointed toby *p.*

Boolean Expressions:

Let us construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

1. $E \rightarrow E_1$ **or** $M\ E_2$
2. | E_1 **and** $M\ E_2$
3. | **not** E_1
4. | (E_1)
5. | **id**$_1$ **relop id**$_2$
6. | **true**
7. | **false**
8. $M \rightarrow \varepsilon$

Synthesized attributes *truelist* and *falselist* of nonterminal *E* are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by *E.truelist* and *E.falselist*.

Consider production $E \rightarrow E_1$ **and** $M\ E_2$. If E_1 is false, then *E* is also false, so the statements on E_1*.falselist* become part of *E.falselist*. If E_1 is true, then we must next test E_2, so the target for thestatements E_1*.truelist* must be the beginning of the code generated for E_2. This target is obtained using marker nonterminal *M*.

Attribute *M.quad* records the number of the first statement of E_2*.code*. With the production M $\rightarrow \varepsilon$ we associate the semantic action

{ *M.quad : = nextquad* }

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the E_1*.truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** ME_2. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* (E_1*.falselist*, *M.quad*);
E.truelist : = *merge*(E_1*.truelist*, E_2*.truelist*);
E.falselist : = E_2*.falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* (E_1*.truelist*, *M.quad*);
E.truelist : = E_2*.truelist*;
E.falselist : = *merge*(E_1*.falselist*, E_2*.falselist*)}

(3) $E \rightarrow$ **not** E_1 { *E.truelist* : = E_1*.falselist*;

E.falselist : = E_1.*truelist*; }

(4) $E \rightarrow (E_1)$ { *E.truelist* : = E_1.*truelist*;
E.falselist : = E_1.*falselist*; }

(5) *E* → **id**$_1$**relop id**$_2$ { *E.truelist* : = *makelist* (*nextquad*);
E.falselist : = *makelist*(*nextquad* + *1*);
emit('if'**id**$_1$.*place* **relop**.*op* **id**$_2$.place '**goto_**')
emit('**goto_**') }

(6) *E* → **true** { *E.truelist* : = *makelist*(*nextquad*);
emit('**goto_**') }

(7) *E* → **false** { *E.falselist* : = *makelist*(*nextquad*);
emit('**goto_**') }

(8) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

1. *S* → **if** *E* **then** *S*
2. | **if***E***then***S***else***S*
3. | **while***E***do***S*
4. | **begin***L***end**
5. | *A*
6. *L* → *L* ; *S*
7. | *S*

Here *S* denotes a statement, *L* a statement list, *A* an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes *E.truelist* and *E.falselist*. *L* and *S* also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L..nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

The semantic rules for the revised grammar are as follows:

E *S* → **if** *E* **then** M_1 S_1 *N* **else** M_2 S_2
{ *backpatch* (*E.truelist*, M_1.*quad*);
backpatch (*E.falselist*, M_2.*quad*);

$S.nextlist := merge\ (S_1.nextlist,\ merge\ (N.nextlist,\ S_2.nextlist))\ \}$

Then backpatch the jumps when *E* is true to the quadruple $M_1.quad$, which is the beginning of the code for S_1. Similarly, we backpatch jumps when *E* is false to go to the beginning of the code for S_2. The list *S.nextlist* includes all jumps out of S_1 and S_2, as well as the jump generated by *N*.

(2) $N \rightarrow \varepsilon$ { *N.nextlist* : = *makelist*(*nextquad*);
emit(‘**goto_**’) }

(3) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

(4) *S* → **if** *E* **then** MS_1 { *backpatch*(*E.truelist*, *M.quad*);
$S.nextlist := merge(\ E.falselist,\ S_1.nextlist)\ \}$

(5) *S* → **while** M_1 *E* **do** M_2 S_1 { *backpatch*($S_1.nextlist$, $M_1.quad$);
backpatch(*E.truelist*, $M_2.quad$);
S.nextlist : = *E.falselist*
emit(‘**goto**’ $M_1.quad$) }

(6) *S* → **begin** *L* **end** { *S.nextlist* : = *L.nextlist* }

(7) *S*→*A* { *S.nextlist* : = **nil** }

The assignment *S.nextlist* : = **nil** initializes *S.nextlist* to an empty list.

(8) *L* → *L1* ; *M S* { *backpatch*($L_1.nextlist$, *M.quad*);
L.nextlist : = *S.nextlist* }

The statement following L_1 in order of execution is the beginning of *S*. Thus the *L1.nextlist* list is backpatched to the beginning of the code for *S*, which is given by *M.quad*.

(9) *L*→*S* { *L.nextlist* : = *S.nextlist* }

3.10 PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement

- *S* → **call id**(*Elist*)
- *Elist* → *Elist* , *E*
- *Elist* → *E*

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence :

- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be generated. For example, consider the following syntax-directed translation

S → **call id**(*Elist*)
{ **for** each item *p* on *queue* **do**
emit (' **param**' *p*);
emit ('call' **id.***place*) }

(2) *Elist* → *Elist* , *E*
{ append *E.place* to the end of *queue* }

(3) *Elist* → *E*
{ initialize *queue* to contain only *E.place* }

- Here, the code for S is the code for *Elist*, which evaluates the arguments, followed by a **param** *p* statement for each argument, followed by a **call** statement.
- *queue* is emptied and then gets a single pointer to the symbol table location for the namethat denotes the value of E.